

Esta clase va a ser

- grabada

Clase 25. PROGRAMACIÓN BACKEND

Proceso principal del servidor + Global & Child process

Temario

24

Segunda práctica integradora

- ✓ Skills
- ✓ Práctica Integradora

25

Proceso principal del servidor + Global & Child process

- ✓ [Process](#)
- ✓ [Manejo de variables de entorno](#)
- ✓ [Global & child process](#)

26

Arquitectura de capas

- ✓ Arquitectura de capas
- ✓ Capas adicionales para Node js
- ✓ Analizando el flujo de arquitectura

Objetivos de la clase

- Conocer el objeto process en nodejs
- Entender las implementaciones principales de un process
- Aprender a manejar variables de entornos

Process

¿Qué es el objeto process?

Cada vez que corremos un proceso en Node.js, éste genera un objeto llamado **process**, el cual contiene información referente a todo lo implicado con el proceso, cosas como:

- ✓ Uso de memoria
- ✓ Id del proceso en el sistema operativo
- ✓ En qué sistema operativo o plataforma está corriendo
- ✓ En qué entorno está corriendo.
- ✓ Qué argumentos tiene el entorno.

Algunos elementos importantes de process

- ✓ `process.cwd()` : Directorio actual del proceso.
- ✓ `process.pid` : id del proceso en el sistema
- ✓ `process.MemoryUsage()` :
- ✓ `process.env` : Accede al objeto del entorno actual
- ✓ `process.argv` : Muestra los argumentos pasados por CLI
- ✓ `process.version` : Muestra la versión del proceso (node en este caso)
- ✓ `process.on()` : Permite setear un listener de eventos
- ✓ `process.exit()` : Permite salir del proceso.

Manejo de argumentos

Argumentos en consola

Los argumentos permiten iniciar la ejecución de un programa a partir de ciertos elementos iniciales. Con argumentos podemos:

- ✓ Setear configuraciones de arranque
- ✓ Agregar valores predeterminados
- ✓ Resolver outputs específicos

Para poder trabajar con argumentos, podemos hacerlo a partir de **process.argv**.



Recordemos que, por defecto, process.argv siempre tendrá dos elementos iniciales

Ejecutando process con diferentes argumentos

JS process.js ×

JS process.js

```
1 // Si ejecutamos "node process.js 1 2 3"
2 console.log(process.argv.slice(2)); // ['1', '2', '3']
3
4 // Si ejecutamos "node process.js a 2 -a"
5 console.log(process.argv.slice(2)); // ['a', '2', '-a']
6
7 // Si ejecutamos "node process.js"
8 console.log(process.argv.slice(2)); // []
9
10 // Si ejecutamos "node process.js --mode development"
11 console.log(process.argv.slice(2)); // ['--mode', 'development']
```

Procesamiento de argumentos con Commander

Commander

<https://www.npmjs.com/package/commander>

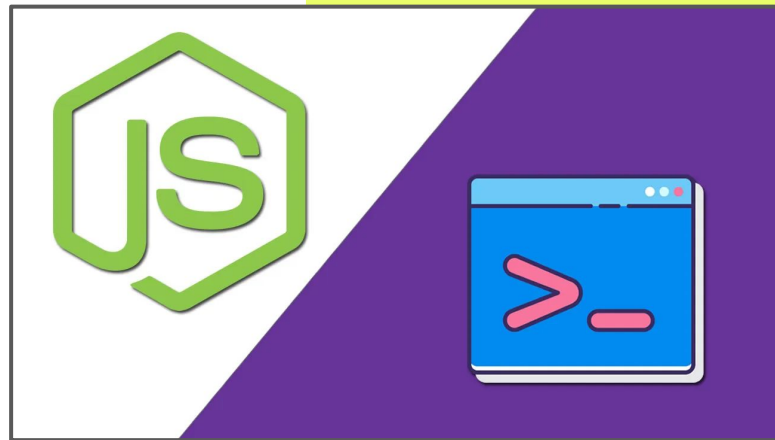
Commander.js es una librería diseñada para facilitar la creación de herramientas de CLI. Permite realizar funciones como:

- ✓ Convertir flags directamente en booleanos
- ✓ Limitar sólo las flags configuradas (cualquier otra impide el procesamiento del programa)
- ✓ Colocar argumentos predeterminados

Y muchas herramientas más para trabajar con argumentos!

Instalamos commander con el comando:

```
npm install commander
```



Ejemplo de uso de commander

JS index.js X

commander > JS index.js > ...

```
1  import { Command } from "commander";
2
3  const program = new Command(); // Inicializamos un programa de commander
4
5  // Especificamos las opciones de nuestro programa
6  program
7    // El 1er argumento es el nombre de la opción, el 2do la descripción y el 3ero el valor por defecto
8    .option("-d", "Variable para debug", false)
9    .option("-p <port>", "Puerto de escucha del servidor", 8080) // <port> es el puerto a ingresar
10   .option("--mode <mode>", "Modo de trabajo", "production") // <mode> es el modo a ingresar
11   .option("-l --letters [letter...]", "Specify letters")
12   // Opción requerida
13   .requiredOption(
14     "-u <user>",
15     "Usuario utilizando la aplicación",
16     "No se ha declarado un usuario"
17   );
18
19  program.parse();
20
21  console.log("Options: ", program.opts());
22  console.log("Remaining arguments: ", program.args);
```

Enviando argumentos a commander

```
→ commander node index.js -d -p 3000 --mode development -u root --letters a b c
Options: {
  d: true,
  p: '3000',
  mode: 'development',
  u: 'root',
  letters: [ 'a', 'b', 'c' ]
}
Remaining arguments: []
```

```
→ commander node index.js -d -p 3000 --mode development -u root 2 a 5 --letters a b c
Options: {
  d: true,
  p: '3000',
  mode: 'development',
  u: 'root',
  letters: [ 'a', 'b', 'c' ]
}
Remaining arguments: [ '2', 'a', '5' ]
```



Para pensar

El manejo de argumentos permite iniciar un programa con valores iniciales

¿Con qué fines consideras que puede utilizarse?

Manejo de variables de entorno

Sobre los entornos



A diagram consisting of a white rectangular box with a thin black border. Inside the box, there are three rounded rectangular buttons stacked vertically. The top button is green and contains the word 'development' in a black, handwritten-style font. The middle button is orange and contains the word 'staging' in the same font. The bottom button is red and contains the word 'production' in the same font.

development

staging

production

Es importante entender que, en un caso real, para que una aplicación esté lista para llegar al cliente, es necesario que pase por diferentes fases, o ser testado en distintos entornos.

Sin embargo, para que estos entornos se encuentren aislados entre sí (no queremos que el entorno de desarrollo tenga datos de producción, o que haya datos de producción en staging), necesitaremos que cada entorno tenga un conjunto de variables específico que determine las características del mismo.

A esto normalmente nos referimos como variables de entorno (environment variables).

Usos de una variable de entorno

El primer uso radica en que una variable cambie según el entorno donde se esté corriendo. Esto permite que, por ejemplo, se pueda apuntar a una base de datos prueba o a una base de datos productiva con sólo cambiar la URL de la base de datos.

Otro factor importante es el factor seguridad. Con las variables de entorno podemos ocultar la información sensible de nuestro código, como credenciales, claves de acceso, tokens, secrets, etc.

dotenv

Utilizando dotenv

<https://www.npmjs.com/package/dotenv>

La librería dotenv nos permitirá setear un entorno en un archivo **.env**, a partir de éste, colocaremos todas las variables que queramos proteger y mantener de manera dinámica.

El archivo no requiere ningún tipo de consideración, bastará con declarar variables de la forma

NOMBRE = VALOR

Para comenzar con dotenv, basta hacer:

```
npm install dotenv
```



Archivo config

Una vez instalado dotenv, crearemos un archivo de configuración llamado **config.js**. En éste, colocaremos un objeto, donde cada par clave/valor corresponderá a la variable de dotenv. todo se basará en utilizar el objeto process nuevamente, esta vez usando

process.env.{VARIABLE}

Usando **dotenv.config()**, indicamos a la computadora que cargue las variables del archivo **.env**, el cual estará seteado dentro del proyecto.

```
JS config.js ×  
JS config.js > ...  
1  import dotenv from "dotenv";  
2  
3  dotenv.config();  
4  
5  export default {  
6    port: process.env.PORT,  
7    mongoUrl: process.env.MONGO_URL,  
8    adminName: process.env.ADMIN_NAME,  
9    adminPassword: process.env.ADMIN_PASSWORD,  
10 };
```

```
⚙ .env ×  
⚙ .env  
1  PORT=8080  
2  MONGO_URL="mongodb://127.0.0.1:27017/my-db"  
3  ADMIN_NAME="adminCoder"  
4  ADMIN_PASSWORD="password"
```

Utilizando la configuración

Una vez finalizada la configuración, podemos importarla y visualizar los resultados:

Al final lo que ocurre es lo siguiente: al ejecutar el comando `dotenv.config()`, éste busca las variables localizadas en `.env` y procede a colocarlas en el objeto `process.env`.

¡Entonces conseguimos tener un objeto de configuración listo para este entorno!

JS index.js

JS index.js

```
1 import config from "./config.js";  
2  
3 console.log(config);  
4
```

→ `dotenv node index.js`

```
{  
  port: '8080',  
  mongoUrl: 'mongodb://127.0.0.1:27017/my-db',  
  adminName: 'adminCoder',  
  adminPassword: 'password'  
}
```

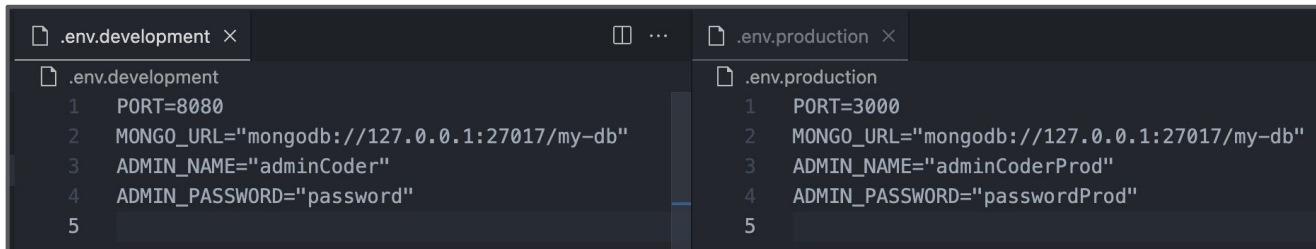
Múltiples entornos

Múltiples entornos = múltiples archivos

Al trabajar con más de un entorno (por ejemplo, desarrollo Y producción), es necesario setear diferentes archivos con el fin de apuntar a diferentes elementos (Por ejemplo, el entorno de desarrollo podría almacenar en archivos, y el entorno productivo podría almacenar en base de datos).

Para esto, setearemos un archivo por cada entorno que deseamos trabajar.

La idea es que ambos archivos tengan las mismas variables, pero con diferentes valores.

 `.env.development` `.env.production`

```
.env.development
1 PORT=8080
2 MONGO_URL="mongodb://127.0.0.1:27017/my-db"
3 ADMIN_NAME="adminCoder"
4 ADMIN_PASSWORD="password"
5

.env.production
1 PORT=3000
2 MONGO_URL="mongodb://127.0.0.1:27017/my-db"
3 ADMIN_NAME="adminCoderProd"
4 ADMIN_PASSWORD="passwordProd"
5
```


Luego, en dotenv...

La ejecución de `dotenv.config()` ya no bastará, ya que esta vez tenemos que poder identificar a cuál archivo queremos apuntar. Para esto utilizaremos la configuración **path**, apuntaremos según lo indicado por la constante "environment".

```
JS config.js ×
JS config.js > ...
1  import dotenv from "dotenv";
2
3  const environment = "production";
4
5  dotenv.config({
6    path:
7      environment === "development" ? "./.env.development" : "./.env.production",
8  });
```

Ejemplo de diferencias según cómo se manda a llamar al entorno

Ejecutando con environment =
'development'

```
{  
  port: '8080',  
  mongoUrl: 'mongodb://127.0.0.1:27017/my-db',  
  adminName: 'adminCoder',  
  adminPassword: 'password'  
}
```

Ejecutando con environment =
'production'

```
{  
  port: '3000',  
  mongoUrl: 'mongodb://127.0.0.1:27017/my-db',  
  adminName: 'adminCoderProd',  
  adminPassword: 'passwordProd'  
}
```



Utilizando argumentos con dotenv

Duración: 10 min



ACTIVIDAD EN CLASE

Utilizando argumentos con dotenv

Consigna:

- ✓ Realizar un servidor basado en node js con express, El cual reciba por flag de cli el comando **--mode <modo>** y sea procesado por commander.
- ✓ Acorde con este argumento, hacer una lectura a los diferentes entornos, y ejecutar dotenv en el path correspondiente a cada modo (--mode development debería conectar con .env.development).
- ✓ Para el entorno development, el servidor debe escuchar en el puerto 8080, para el entorno productivo, el servidor debe escuchar en el puerto 3000.



Break

¡10 minutos y volvemos!

Continuando con process

Global & child process

Listeners

Además de todas las funcionalidades que hemos visto de process, tenemos otra más para trabajar: el método **on**. Permitirá poner a nuestro proceso principal a la escucha de algún evento para poder ejecutar alguna acción en caso de que algo ocurra.

Algunos de los listeners más utilizados son

- ✓ on 'exit' : Para ejecutar un código **justo antes** de la finalización del proceso.
- ✓ on 'uncaughtException' : Para atrapar alguna excepción que no haya sido considerada en algún catch
- ✓ on 'message' para poder comunicarse con otro proceso

listeners con process.on

```
process.on("exit", (code) => {  
  console.log("Este código será ejecutado antes de salir del proceso");  
});  
  
process.on("uncaughtException", (err, origin) => {  
  console.log("Este código atraparé cualquier excepción no capturada");  
});  
  
process.on("message", (message) => {  
  console.log(  
    "Este código se ejecutará cuando se reciba un mensaje de otro proceso"  
  );  
});
```


Por ejemplo, sin en algún punto llegase a cometer algún error y el mismo no sea controlado:

```
throw new Error("Random error");
```

Este código atraparé cualquier excepción no capturada
Este código será ejecutado antes de salir del proceso

Códigos de salida de process

Cuando ejecutamos una salida con `process.exit()` como argumento, podemos enviar un código que sirve como identificador para el desarrollador sobre la razón de la salida

Hay que conocer los códigos de salida para saber cómo utilizarlos. También podemos crear nuestros propios códigos.

Algunos de los códigos importantes son:

0: proceso finalizado normalmente.

1: proceso finalizado por excepción fatal

5: Error fatal del motor V8.

9: Para argumentos inválidos al momento de la ejecución.



Ejemplo en vivo

- ✓ Se creará una función llamada "listNumbers" el cual recibirá un número indefinido de argumentos (...numbers)
- ✓ Si se pasa un argumento no numérico, entonces deberá mostrar por consola un error indicando "Invalid parameters" seguido de una lista con los tipos de dato (para [1,2,"a",true], el error mostrará [number,number,string,boolean])
- ✓ Escapar del proceso con un código -4. Utilizando un listener, obtener el código de escape del error y mostrar un mensaje "Proceso finalizado por argumentación inválida en una función"

Duración: **15 min**

Child process

Un proceso creando otro proceso

Algunas operaciones requieren mucho procesamiento, como:

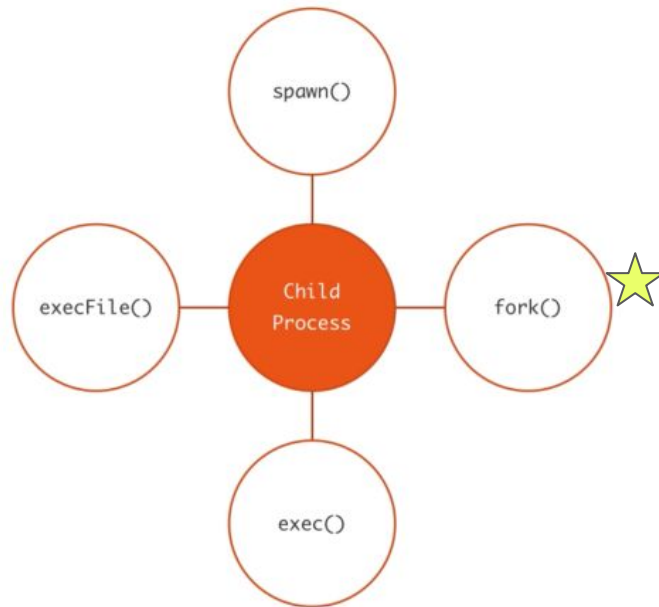
- ✓ Lectura de archivos enormes
- ✓ Consultas a bases muy complejas

En casos como estos, tenemos la posibilidad de, a partir de un proceso de Node, crear otro sub-proceso para poder resolver estas tareas. De esta manera, en nuestro caso evitaremos bloquear las tareas principales de un servidor (responder peticiones).

Cómo crear un child process

Existen diferentes formas para que un proceso de Node pueda ejecutar otro proceso: hay cuatro operadores que pueden ser utilizados y manipulados de diferentes formas

En esta clase aprenderemos sobre el método [`fork\(\)`](#). Sin embargo, se te invita a que profundices sobre los diferentes métodos e indagues contextos de aplicación.





método fork(): problema

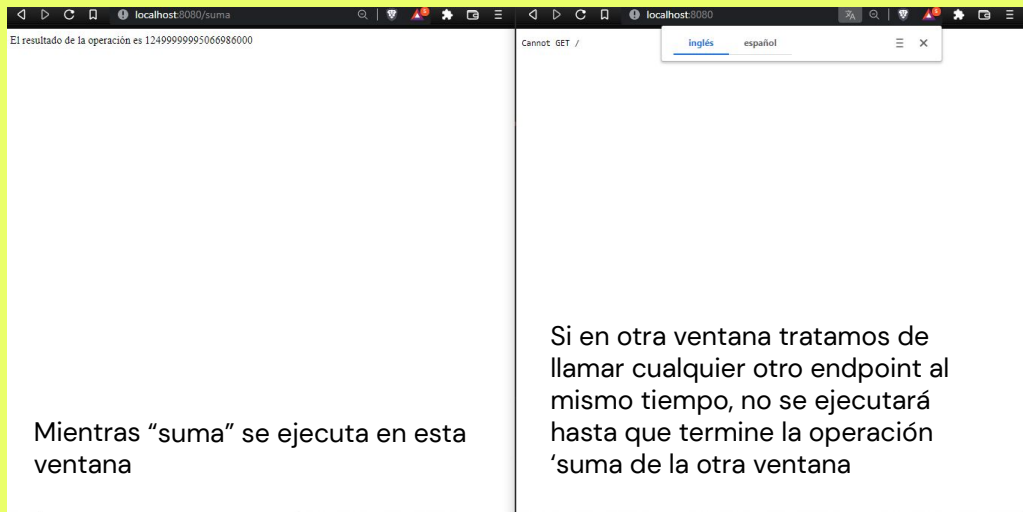
Caso de uso de fork

Supongamos que tenemos una operación en nuestro servidor express que consume muchos recursos y demora una fuerte cantidad de tiempo. Por ejemplo, una suma que se realiza a lo largo de un gran número de iteraciones ($5e9 = 5000000000$)

```
function operacionCompleja() {  
  let result = 0;  
  
  for (let i = 0; i < 5e9; i++) {  
    result += i;  
  }  
  
  return result;  
}  
  
app.get("/suma", (req, res) => {  
  const result = operacionCompleja();  
  
  res.send(`El resultado es ${result}`);  
});
```



método fork: problema



Comienzan las complicaciones

Al llamar al endpoint '/suma', notamos algo horrible! Las operaciones se han bloqueado en todos los demás endpoints si se llaman en paralelo.

Pensando una solución

No podemos simplemente “no hacer la operación”. Debemos buscar la manera de que esta operación se pueda hacer sin bloquear otros endpoints. 🤔

¿Y si delegamos a un proceso hijo para que sólo se encargue de ese proceso, mientras yo sigo atendiendo solicitudes? Suena bien ¿no?



Comenzando a estructurar un forkeo

1. Separar la función que está causando problemas en un archivo diferente y reestructurarla para que sólo se ejecute cuando el padre se lo indique (usaremos el `process.on('message')`)

JS operacionCompleja.js ×

JS operacionCompleja.js > ...

```
1 function operacionCompleja() {
2   let result = 0;
3
4   for (let i = 0; i < 5e9; i++) {
5     result += i;
6   }
7
8   return result;
9 }
```

JS operacionCompleja.js ×

JS operacionCompleja.js > ...

```
1 process.on("message", (message) => {
2   let result = 0;
3
4   for (let i = 0; i < 5e9; i++) {
5     result += i;
6   }
7
8   process.send(result);
9 });
```

el return se convierte en `process.send`



Volviendo a nuestro app.js

Comenzamos importando fork:

```
import { fork } from "child_process";
```

No es necesario instalar nada ya que el módulo "child_process" es un módulo nativo. Luego, reestructuramos nuestro endpoint /suma

```
app.get("/suma", (req, res) => {  
  const child = fork("./operacionCompleja.js"); // Forkeamos la operación  
  
  child.send("start"); // Damos inicio ("start" podría ser cualquier mensaje en este caso)  
  
  // Escuchamos el mensaje que nos envía el proceso hijo  
  child.on("message", (result) => {  
    res.send(`El resultado es ${result}`);  
  });  
});
```

Recapitulando...

- El padre realiza un **fork** al proceso hijo.
- El padre **envía un mensaje** al proceso hijo
- El proceso hijo **tiene su propio listener**, al recibir el mensaje del padre, entiende que tiene que comenzar con su cálculo.
- Una vez que el hijo termina de calcular, le **envía un mensaje** al padre, donde el contenido del mensaje será el resultado.
- Ese resultado se envía al cliente.



Cálculo bloqueante con contador

Duración: 15 min



ACTIVIDAD EN CLASE

Cálculo bloqueante con contador

Realizar un servidor en express que contenga una ruta raíz '/' donde se represente la cantidad de visitas totales a este endpoint

Se implementará otra ruta '/calculo-bloq', que permita realizar una suma incremental de los números del 0 al 5000000000 con el siguiente algoritmo.

```
function sumar() {  
  let suma = 0;  
  for (let i = 0; i < 1e9; i++) {  
    suma += i;  
  }  
  
  return suma;  
}
```



ACTIVIDAD EN CLASE

Cálculo bloqueante con contador

Comprobar que al alcanzar esta ruta en una pestaña del navegador, el proceso queda en espera del resultado. Constatar que durante dicha espera, la ruta de visitas no responde hasta terminar este proceso.

Luego crear la ruta '/calculo-nobloq' que hará dicho cálculo forkeando el algoritmo en un `child_process`, comprobando ahora que el request a esta ruta no bloquee la ruta de visitas.

¿Preguntas?

Opina y valora
esta clase

Resumen

de la clase hoy

- ✓ Process
- ✓ Manejo de variables de entorno
- ✓ Global & child process

Muchas gracias.

#DemocratizandoLaEducación